ECE 4750 Computer Architecture, Fall 2016
Verilog Usage Rules

School of Electrical and Computer Engineering
Cornell University

revision: 2016-11-05-14-20

Verilog is a powerful language that was originally intended for building simulators of hardware as opposed to models that could automatically be transformed into hardware (e.g., synthesized to an FPGA or ASIC). Given this, it is very easy to write Verilog code that does not actually model any kind of realistic hardware. Indeed, we actually need this feature to be able to write clean and productive assertions and line tracing. Non-synthesizable Verilog modeling is also critical when implementing test harnesses. **So students must be very diligent in actively deciding whether or not they are writing synthesizable register-transfer-level models or non-synthesizable code. Students must always keep in mind what hardware they are modeling and how they are modeling it!**

Students' design work will almost exclusively use synthesizable register-transfer-level (RTL) models. It is acceptable to include a limited amount of non-synthesizable code in students' designs for the sole purpose of debugging, assertions, or line tracing. If the student includes non-synthesizable code in their actual design (i.e., not the test harness), they must explicitly demarcate this code by wrapping it in `‘ifndef SYNTHESIS` and `‘endif`. This explicitly documents the code as non-synthesizable and aids automated tools in removing this code before synthesizing the design. **If at any time students are unclear about whether a specific construct is allowed in a synthesizable RTL, they should ask the instructors.**

The next page includes a table that outlines which Verilog constructs are allowed in synthesizable RTL, which constructs are allowed in synthesizable RTL with limitations, and which constructs are explicitly not allowed in synthesizable RTL. There are no limits on using the Verilog preprocessor, since the preprocessing step happens at compile time.

<table>
<tr><th>Always Allowed in Synthesizable RTL</th><th>Allowed in Synthesizable RTL With Limitations</th><th>Explicitly Not Allowed in Synthesizable RTL</th></tr>
<tr><td>logic</td><td>always[1]</td><td>wire, reg[13]</td></tr>
<tr><td>logic [N-1:0]</td><td>enum[2]</td><td>integer, real, time, realtime</td></tr>
<tr><td>& | ^ ^~ ~ (bitwise)</td><td>struct[3]</td><td>signed[14]</td></tr>
<tr><td>&& || !</td><td>casez, endcase[4]</td><td>===, !==</td></tr>
<tr><td>& ~& | ~| ^ ^~ (reduction)</td><td>task, endtask[5]</td><td>* / % ** [15]</td></tr>
<tr><td>+ -</td><td>function, endfunction[5]</td><td>#N (delay statements)</td></tr>
<tr><td>>> << >>></td><td>= (blocking assignment)[6]</td><td>inout[16]</td></tr>
<tr><td>== != > <= < <=</td><td><= (non-blocking assignment)[7]</td><td>initial</td></tr>
<tr><td>{}</td><td>typedef[8]</td><td>variable initialization[17]</td></tr>
<tr><td>{N{}} (repeat)</td><td>packed[9]</td><td>negedge[18]</td></tr>
<tr><td>?:</td><td>$clog2()[10]</td><td>casex, endcase</td></tr>
<tr><td>always_ff, always_comb</td><td>$bits()[10]</td><td>for, while, repeat, forever[19]</td></tr>
<tr><td>if else</td><td>$signed()[11]</td><td>fork, join</td></tr>
<tr><td>case, endcase</td><td>read-modify-write signal[12]</td><td>deassign, force, release</td></tr>
<tr><td>begin, end</td><td></td><td>specify, endspecify</td></tr>
<tr><td>module, endmodule</td><td></td><td>nmos, pmos, cmos</td></tr>
<tr><td>input, output</td><td></td><td>rnmos, rpmos, rcmos</td></tr>
<tr><td>assign</td><td></td><td>tran, tranif0, tranif1</td></tr>
<tr><td>parameter</td><td></td><td>rtran, rtranif0, rtranif1</td></tr>
<tr><td>localparam</td><td></td><td>supply0, supply1</td></tr>
<tr><td>genvar</td><td></td><td>strong0, strong1</td></tr>
<tr><td>generate, endgenerate</td><td></td><td>weak0, weak1</td></tr>
<tr><td>generate for</td><td></td><td>primitive, endprimitive</td></tr>
<tr><td>generate if else</td><td></td><td>defparam</td></tr>
<tr><td>generate case</td><td></td><td>unnamed port connections[20]</td></tr>
<tr><td>named port connections</td><td></td><td>unnamed parameter passing[21]</td></tr>
<tr><td>named parameter passing</td><td></td><td>all other keywords</td></tr>
<tr><td></td><td></td><td>all other system tasks</td></tr>
</table>

1 Students should prefer using `always_ff` and `always_comb` instead of `always`. If students insist on using `always`, then it can only be used in one of the following two constructs: `always @(posedge clk)` for sequential logic, and `always @(*)` for combinational logic. Students are not allowed to trigger sequential blocks off of the negative edge of the clock or create asynchronous resets, nor use explicit sensitivity lists.

2 `enum` can only be used with an explicit base type of `logic` and explicitly setting the bitwidth using the following syntax: `typedef enum logic [$clog2(N)-1:0] { ... } type_t;` where `N` is the number of labels in the `enum`. Anonymous `enum`s are not allowed.

3 `struct` can only be used with the `packed` qualifier (i.e., `unpacked` structs are not allowed) using the following syntax: `typedef struct packed { ... } type_t;` Anonymous `struct`s are not allowed.

4 `casez` can only be used in very specific situations to compactly implement priority encoder style hardware structures.

5 `task` and `function` blocks must themselves contain only synthesizable RTL.

6 Blocking assignments should only be used in `always_comb` blocks and are explicitly not allowed in `always_ff` blocks.

7 Non-blocking assignments should only be used in `always_ff` blocks and are explicitly not allowed in `always_comb` blocks.

8 `typedef` should only be used in conjunction with `enum` and `struct`.

9 `packed` should only be used in conjunction with `struct`.

10 The input to `$clog2/$bits` must be a static-elaboration-time constant. The input to `$clog2/$bits` cannot be a signal (i.e., a wire or a port). In other words, `$clog2/$bits` can only be used for static elaboration and cannot be used to model actual hardware.

11 `$signed()` can only be used around the operands to `>>>`, `>`, `>=`, `<`, `<=` to ensure that these operators perform the signed equivalents.

12 Reading a signal, performing some arithmetic on the corresponding value, and then writing this value back to the same signal (i.e., read-modify-write) is not allowed within an `always_comb` concurrent block. This is a combinational loop and does not model valid hardware. Read-modify-write is allowed in an `always_ff` concurrent block with a non-blocking assignment, although we urge students to consider separating the sequential and combinational logic. Students can use an `always_comb` concurrent block to read the signal, perform some arithmetic on the corresponding value, and then write a temporary wire; and use an `always_ff` concurrent block to flop the temporary wire into the destination signal.

13 `wire` and `reg` are perfectly valid, synthesizable constructs, but `logic` is much cleaner. So we would like students to avoid using `wire` and `reg`.

14 `signed` types can sometimes be synthesized, but we do not allow this construct in the course.

15 Multipliers can be synthesized and small multipliers can even be synthesized efficiently, but we do not allow this construct in the course. If you need to multiply or divide by a power of two, then you can use the left and right shift operators.

16 Ports with `inout` can be used to create tri-state buses, but tools often have trouble synthesizing hardware from these kinds of models.

17 Variable initialization means assigning an initial value to a `logic` variable when you declare the variable. This is not synthesizable; it is not modeling real hardware. If you need to set some state to an initial condition, you must explicitly use the `reset` signal.

18 Triggering a sequential block off of the `negedge` of a signal is certainly synthesizable, but we will be exclusively using a positive-edge-triggered flip-flop-based design style.

19 If you would like to generate hardware using loops, then you should use `generate` blocks.

20 In very specific, rare cases unnamed port connections might make sense, usually when there are just one or two ports and there purpose is obvious from the context.

21 In very specific, rare cases unnamed parameter passing might make sense, usually when there are just one or two parameters and their purpose is obvious from the context.